Developing IceFIX Applications

Previous Next

This page show how to use the IceFIX Slice API to develop a client in C#.

On this page:

What is an IceFIX client?

An IceFIX client is an Ice client that uses a FIX library (such as QuickFIX) to encode FIX messages and send them to an acceptor via an IceFIX Bridge. Similarly, the client must decode FIX messages that the bridge relays from the acceptor. Unlike a regular FIX initiator, an IceFIX client requires no logic for handling FIX sessions, nor is it responsible for managing the connection to a FIX acceptor.

Registering a client

During installation, a client must register with the bridge using a unique identity and a requested quality of service. If the client does not already have a unique identity, it can obtain one from the bridge. After registration, the bridge stores and forwards any FIX messages for the client.

A client calls `BridgeAdmin::register` to perform the initial registration:

C#

```
IceFIX.BridgePrx bridge = ...;
IceFIX.BridgeAdminPrx admin = bridge.getAdmin();
Dictionary<string, string> qos = new Dictionary<string, string>();
string id = admin.register(qos);
```

The call to `register` returns a unique identity assigned by the bridge, which the client must store for future use. If the client already has a unique identity, it must register using `BridgeAdmin::registerWithId` instead:

C#

```
IceFIX.BridgePrx bridge = ...;
string id = ...;
IceFIX.BridgeAdminPrx admin = bridge.getAdmin();
Dictionary<string, string> qos = new Dictionary<string, string>();
admin.registerWithId(id, qos);
```

When uninstalled, the client should also unregister from the bridge:

C#

```
IceFIX.BridgePrx bridge = ...;
string id = ...;
IceFIX.BridgeAdminPrx admin = bridge.getAdmin();
admin.unregister(id, false);
```

The bridge discards any future messages for this client.

Back to Top ^

Connecting to the bridge

In order to send and receive FIX-encoded messages, the client must connect to the bridge using its registered identity:

C#

```
IceFIX.BridgePrx bridge = ...;
string id = ...;
IceFIX.ReporterPrx reporter = ...;
IceFIX.ExecutorPrx executor = bridge.connect(id, reporter);
```

The `Bridge::connect` operation requires the client to supply a proxy for a `Reporter` object, which the bridge uses to deliver FIX messages to the client. The `connect` operation returns a proxy for an `Executor` object that the client uses to send FIX messages.

Back to Top ^

Using an executor

A client uses the executor object to send messages to the FIX acceptor:

C#

```
IceFIX.ExecutorPrx = ...;
QuickFIX42.NewOrderSingle req = new QuickFIX42.NewOrderSingle(
    clOrdID, handleInst, symbol, side, transatTime, ordType);
req.set(price);
req.set(orderQty);
req.set(timeInForce);
int seqNum = executor.execute(req.ToString());
```

In this example, we compose the FIX message using the QuickFIX .NET library. The code converts the `NewOrderSingle` FIX message to its encoded representation using `ToString` and then invokes `execute` to send the message to the bridge. Note that it is not necessary to set the `SenderCompID` or `TargetCompID` fields as would be necessary in a regular QuickFIX application; since the IceFIX bridge is associated with only one acceptor, the bridge supplies this information.

Once the client has completed its work, it should call `destroy` on the executor object to inform the bridge that the client is no longer active. The bridge stores any subsequent messages and forwards them when the client reconnects.

Back to Top ^

Implementing a reporter

The IceFIX bridge calls back on a `Reporter` object to deliver FIX messages for the client:

C#

```
public class ReporterI : IceFIX.ReporterDisp_
{
    public override void message(string data, Ice.Current current)
    {
        // ...
    }
}
```

The bridge invokes `message` for each FIX message. The `data` parameter contains the FIX-encoded representation of the message.

In order to process the message, the client must first convert the string into a QuickFIX message. The following code converts a FIX 4.2 message:

C#

```
Message message = new Message(data);
string msgType = message.getHeader().getField(MsgType.FIELD);
QuickFIX.MessageFactory factory = new QuickFIX42.MessageFactory();
Message msg = factory.create(data, msgType);
msg.setstring(data);
```

Now that we have created the QuickFIX message type, our next step is to decode the message. Typically this is done using a QuickFIX `MessageCracker`. A normal QuickFIX application might be implemented as follows:

C#

```
public class Application : MessageCracker, QuickFIX.Application
{
    // Other on* events
    public void fromApp(Message message, SessionID sessionID)
    {
        crack( message, sessionID );
    }

    public override void onMessage(QuickFIX42.ExecutionReport r, SessionID id)
    {
        // Process report
    }
}
```

An IceFIX client uses a similar approach:

C#

```
public class Application : QuickFIX42.MessageCracker
{
    public override void onMessage(QuickFIX42.ExecutionReport r, SessionID id)
    {
        // Process report
    }
}

public class ReporterI : IceFIX.ReporterDisp_
{
    Application app;

    ReporterI(Application a)
    {
        app = a;
    }

    public override void message(string data, Ice.Current current)
    {
        // ...
        SessionID id = new SessionID();
        app.crack(msg, id};
    }
}
```

We are now ready to connect to the bridge. We need to create an object adapter for our `Reporter` servant and obtain the proxy required by the `connect` operation:

C#

```
IceFIX.BridgePrx bridge = ...;
string id = ...;
Application app = ...;
Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Client");
IceFIX.ReporterPrx reporter = IceFIX.ReporterPrxHelper.uncheckedCast(
    adapter.addWithUUID(new ReporterI(app)));
adapter.activate();
IceFIX.ExecutorPrx executor = bridge.connect(id, reporter);
```

We can also use an alternative implementation strategy in which our servant implements both the `MessageCracker` interface as well as the `Reporter` Ice object with the help of Tie classes. Tie classes allow us to implement a servant using delegation instead of inheritance:

C#

```
public class ReporterI : QuickFIX42.MessageCracker, IceFIX.ReporterOperations_
{
    public void message(string data, Ice.Current current)
    {
        Message msg = ...; // Create the QuickFIX message
        SessionID id = new SessionID();
        crack(msg, id);
    }

    public override void onMessage(QuickFIX42.ExecutionReport r, SessionID id)
    {
        // Process report
    }
}

IceFIX.BridgePrx bridge = ...;
string id = ...;
Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Client");
IceFIX.ReporterPrx reporter = IceFIX.ReporterPrxHelper.uncheckedCast(
    adapter.addWithUUID(new ReporterTie_(new ReporterI())));
adapter.activate();
IceFIX.ExecutorPrx executor = bridge.connect(id, reporter);
```

Back to Top ^

Using multiple bridges

If a client connects to multiple bridges, it must register separately with each bridge. The client may register using the same identity for each bridge as long as no other bridge client uses that identity.

The client may also register the same `Reporter` object with multiple bridges, but in some circumstances your `Reporter` object may need the ability to determine which bridge sent a FIX message. In this case, you can use a variety of techniques to determine the originating bridge. The simplest technique is to use a `Reporter` servant per bridge.

A more complex solution uses only one servant but multiple Ice objects. You can discover the originating bridge in this case by assigning a unique identity to each reporter object and examining its identity in the `message` method. For example, let's assume a client connects to two bridges named `TP1` and `TP2`:

C#

```
Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Client");
adapter.activate();
Ice.Object servant = new ReporterTie_(new ReporterI()));

IceFIX.BridgePrx tp1 = ...;
string tp1Id = ...;
Ice.Identity tp1Identity = new Ice.Identity();
tp1Identity.category = "TP1";
tp1Identity.name = Ice.Util.generateUUID();
IceFIX.ReporterPrx tp1Reporter = IceFIX.ReporterPrxHelper.uncheckedCast(
    adapter.add(servant, tp1Identity));
IceFIX.ExecutorPrx tp1Executor = bridge.connect(tp1id, tp1Reporter);

IceFIX.BridgePrx tp2 = ...;
string tp2Id = ...;
Ice.Identity tp2Identity = new Ice.Identity();
tp2Identity.category = "TP2";
tp2Identity.name = Ice.Util.generateUUID();
IceFIX.ReporterPrx tp2Reporter = IceFIX.ReporterPrxHelper.uncheckedCast(
    adapter.add(servant, tp2Identity));
IceFIX.ExecutorPrx tp2Executor = bridge.connect(tp2id, tp2Reporter);
```

Here we create a single servant and register it with the object adapter under two identities: `TP1/<uuid>` and `TP2/<uuid>`. In the servant implementation, we can determine the calling bridge as follows:

C#

```
public class ReporterI : IceFIX.ReporterDisp_
{
    public override void message(string data, Ice.Current current)
    {
        if(current.id.category.Equals("TP1"))
        {
            // TP1 sent the message.
        }
        else if(current.id.category.Equals("TP2"))
        {
            // TP2 sent the message.
        }
    }
}
```

Back to Top ^

Qualities of Service

The IceFIX bridge supports a single quality of service parameter named `filtered`. If a client sets this parameter to true, the bridge only delivers to a client the FIX messages that are replies to the client's requests. If the parameter is false, the client receives every FIX message sent to the bridge. If the `filtered` parameter is not defined, the default value is true.

A client specifies its desired quality of service during registration, as shown in the following example:

C#

```
IceFIX.BridgePrx bridge = ...;
IceFIX.BridgeAdminPrx admin = bridge.getAdmin();
Dictionary<string, string> qos = new Dictionary<string, string>();
qos["filtered"] = "false";
string id = admin.register(qos);
```

Back to Top ^

Previous

Next